Hacking the Query Planner, Again

Richard Guo / VMware
PGCon 2020

Agenda

- **What does planner do?**
- Phases of planning

Overall backend structure

- Parser
  - Determines the semantic meaning of a query string
- Rewriter
  - Performs view and rule expansion
- Planner
  - Designs an execution plan for the query
- Executor
  - Runs the plan

What does planner do?

- For a given query, find a correct execution plan that has the lowest "cost"
  - A given query can be actually executed in a wide variety of different ways
  - If it is computationally feasible, examine each of these possible ways, represented by data structures called **Path**
  - Select the cheapest Path and convert it to a full-fledged **Plan**

Agenda

- What does planner do?
- **Phases of planning**

Phases of planning

- Preprocessing
  - simplify the query if possible; collect information
- Scan/join planning
  - decide how to implement FROM/WHERE
- Post scan/join planning
  - deal with plan steps that aren’t scans or joins
- Postprocessing
  - convert results into form the executor wants

Early preprocessing

- Simplify scalar expressions
- Expand simple SQL functions in-line
- Simplify join tree

Simplify scalar expressions

- Simplify function calls
  - The function is strict and has any constant-null inputs

    ```
    int4eq(1,NULL) => NULL
    ```

  - The function is immutable and has all constant inputs

    ```
    2 + 2 => 4
    ```

Simplify scalar expressions

- Simplify boolean expressions

```
"x OR true" => "true"

"x AND false" => "false"
```

Simplify scalar expressions

- Simplify CASE expressions

```
CASE WHEN 2+2 = 4 THEN x+1

ELSE 1/0 END

     ⇒ x+1

     ... not “ERROR: division by zero”
```

Why bother simplifying?

- Do computations only once, not once per row
- Exploit constant-folding opportunities exposed by view expansion and SQL function inlining

Expand simple SQL functions in-line

```
CREATE FUNCTION incr4(int) RETURNS int

AS 'SELECT $1 + (2 + 2)' LANGUAGE SQL;



SELECT incr4(a) FROM foo;

=>

SELECT a + 4 FROM foo;
```

Why bother inlining SQL functions?

- Avoid the rather high per-call overhead of SQL functions
- Expose opportunities for constant-folding within the function expression

Simplify join tree

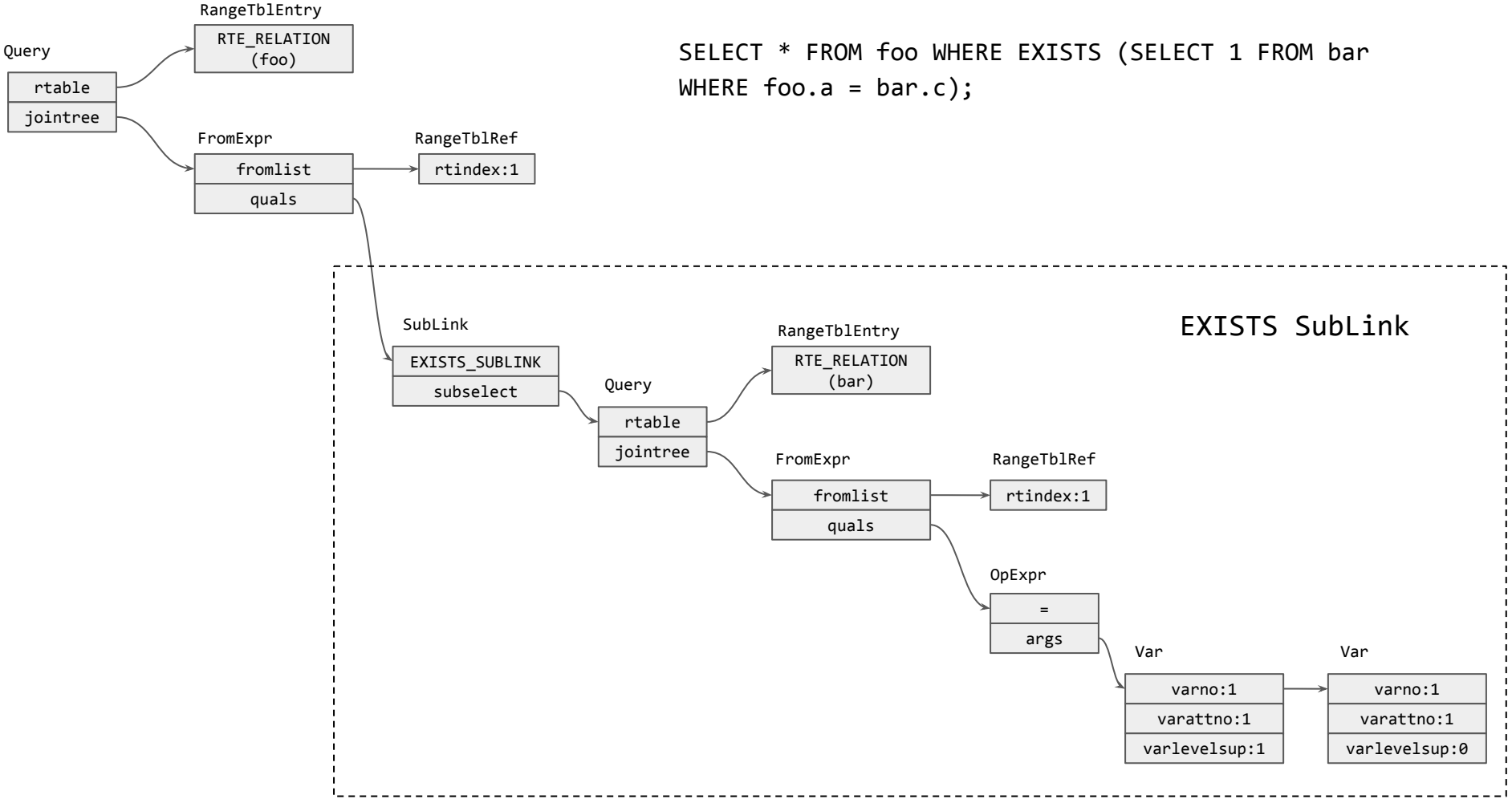
- Convert IN, EXISTS sub-selects to semi-joins
- Flatten (“pull up”) sub-selects if possible
- Flatten UNION ALL, expand inheritance trees
- Reduce outer joins to inner joins
- Reduce outer joins to anti joins

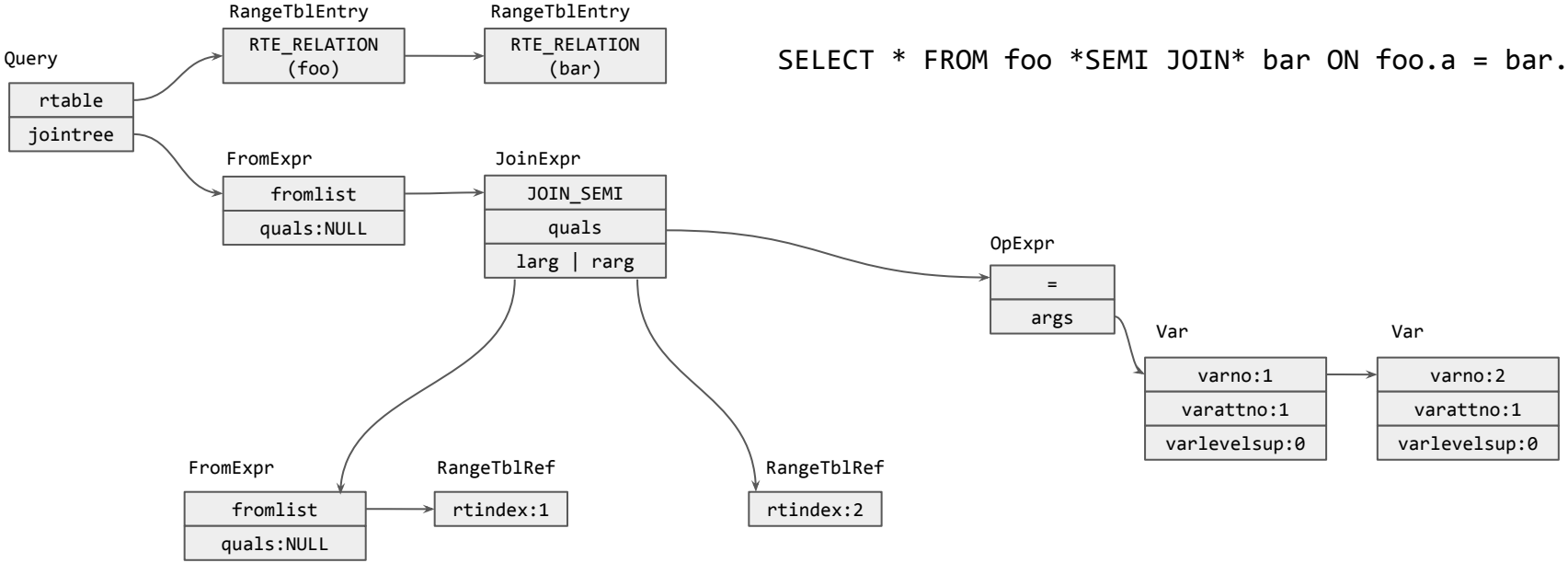
Convert IN, EXISTS sub-selects to semi-joins

```
SELECT * FROM foo WHERE EXISTS (SELECT 1 FROM bar WHERE foo.a = bar.c);

=>

SELECT * FROM foo *SEMI JOIN* bar ON foo.a = bar.c;
```


SELECT * FROM foo WHERE EXISTS (SELECT 1 FROM bar WHERE foo.a = bar.c);
Query
rtable
jointree
RangeTblEntry
RTE_RELATION (foo)
FromExpr
fromlist
quals
RangeTblRef
rtindex:1
EXISTS SubLink
SubLink
EXISTS_SUBLINK
subselect
Query
rtable
jointree
RangeTblEntry
RTE_RELATION (bar)
FromExpr
fromlist
quals
RangeTblRef
rtindex:1
OpExpr
=
args
Var
varno:1
varattno:1
varlevelsup:1
Var
varno:1
varattno:1
varlevelsup:0



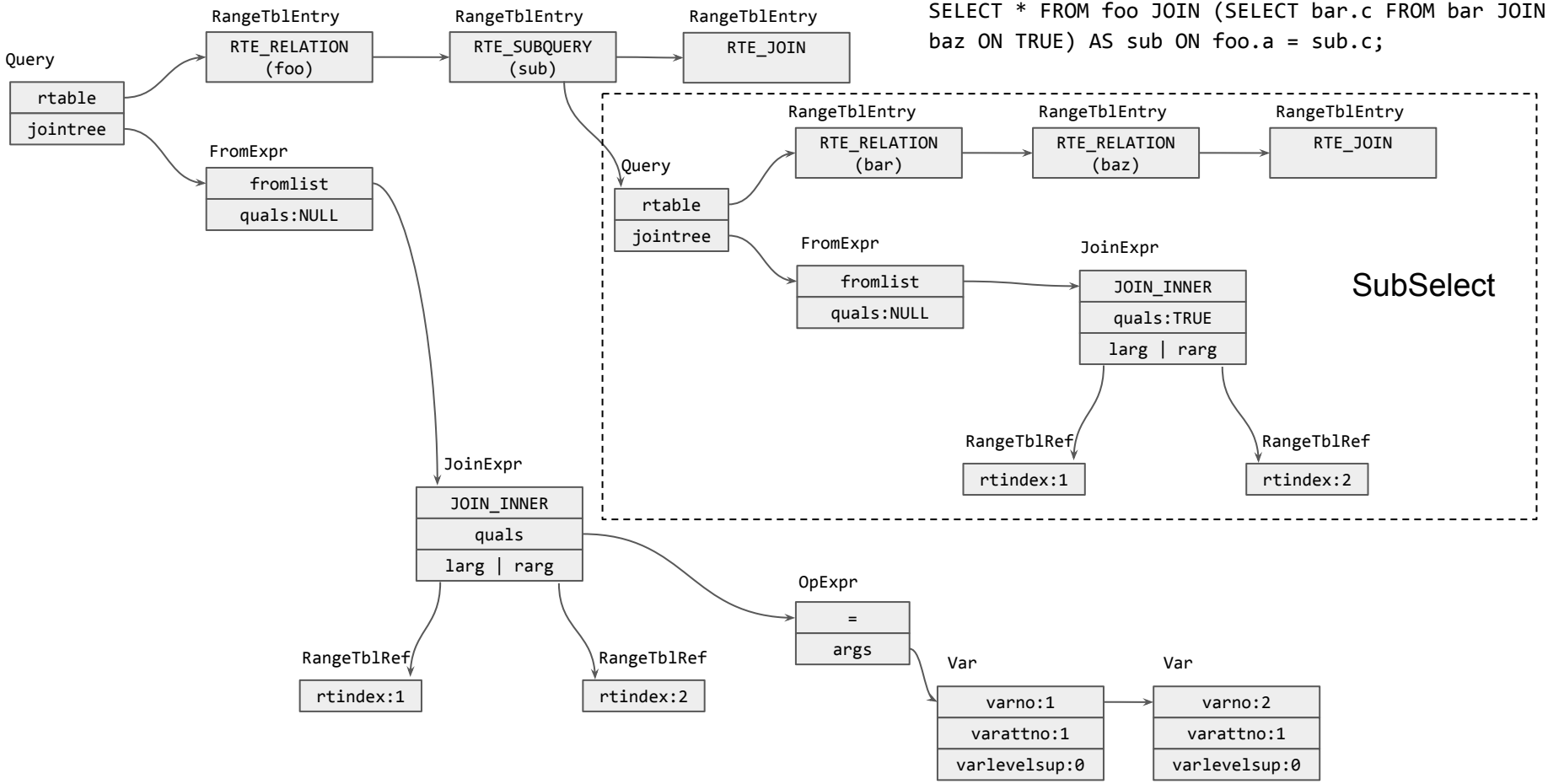

```
SELECT * FROM foo *SEMI JOIN* bar ON foo.a = bar.c;
```

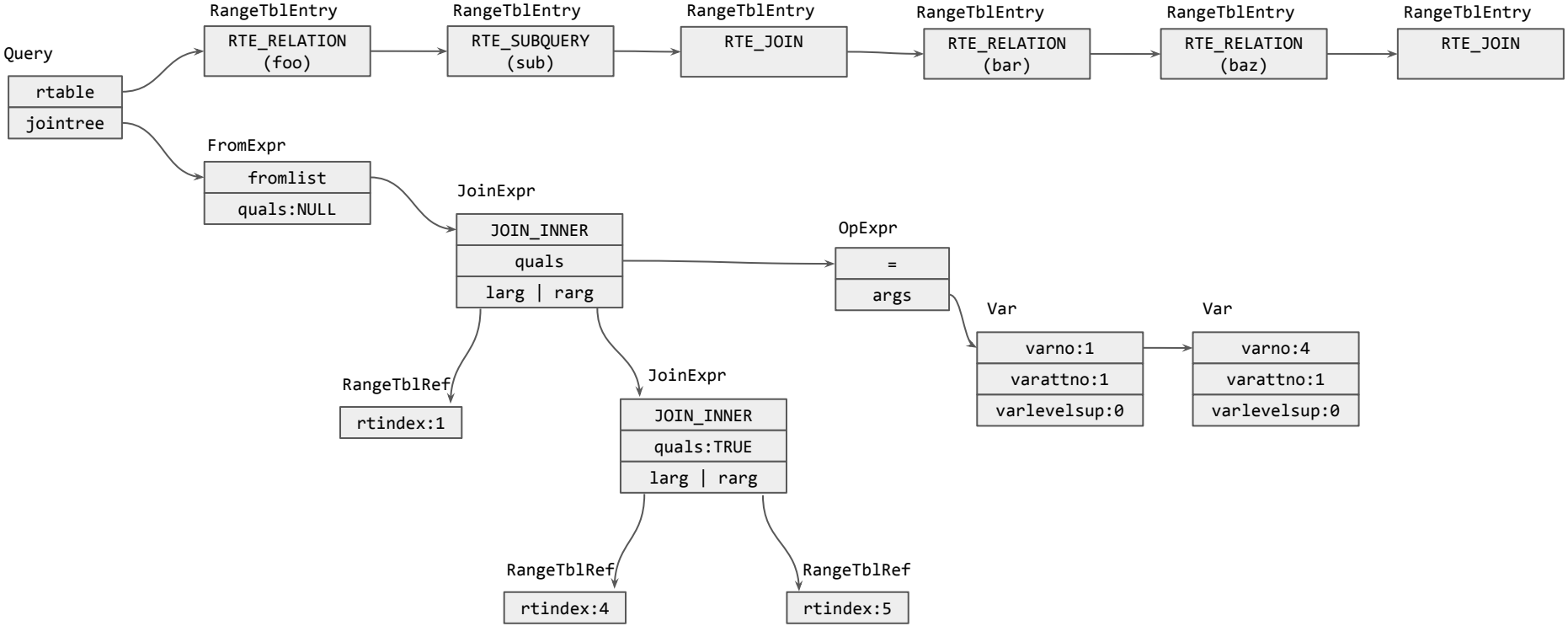
Flatten (“pull up”) sub-selects if possible

```
SELECT * FROM foo JOIN (SELECT bar.c FROM bar JOIN baz ON TRUE) AS sub ON foo.a =
sub.c;

=>

SELECT * FROM foo JOIN (bar JOIN baz ON TRUE) ON foo.a = bar.c;
```


SELECT * FROM foo JOIN (SELECT bar.c FROM bar JOIN baz ON TRUE) AS sub ON foo.a = sub.c;
Query
rtable
jointree
RangeTblEntry
RTE_RELATION (foo)
RangeTblEntry
RTE_SUBQUERY (sub)
RangeTblEntry
RTE_JOIN
FromExpr
fromlist
quals:NULL
JoinExpr
JOIN_INNER
quals
larg | rarg
RangeTblRef
rtindex:1
RangeTblRef
rtindex:2
OpExpr
=
args
Var
varno:1
varattno:1
varlevelsup:0
Var
varno:2
varattno:1
varlevelsup:0
Query
rtable
jointree
RangeTblEntry
RTE_RELATION (bar)
RangeTblEntry
RTE_RELATION (baz)
RangeTblEntry
RTE_JOIN
FromExpr
fromlist
quals:NULL
JoinExpr
JOIN_INNER
quals:TRUE
larg | rarg
RangeTblRef
rtindex:1
RangeTblRef
rtindex:2
SubSelect




```
SELECT * FROM foo JOIN (bar JOIN baz ON TRUE) ON
foo.a = bar.c;
```

Why bother flattening sub-selects?

- It may help produce a better plan to pull up a subquery into the parent query and consider it as part of the entire plan search space
- Otherwise the subquery would be planned independently and treated as a "black box" during planning of the outer query

Reduce outer joins to inner joins

- If there is a strict qual above the outer join that constrains a Var from the nullable side of the join to be non-null

```
SELECT ... FROM foo LEFT JOIN bar ON (...) WHERE bar.d = 42;

=>

SELECT ... FROM foo INNER JOIN bar ON (...) WHERE bar.d = 42;
```

Reduce outer joins to anti joins

- If the outer join's own quals are strict for any nullable Var that was forced null by higher qual levels

```
SELECT * FROM foo LEFT JOIN bar ON foo.a = bar.c WHERE bar.c IS NULL;

=>

SELECT * FROM foo *ANTI JOIN* bar on foo.a = bar.c;
```

Later preprocessing

- Distribute WHERE and JOIN/ON qual clauses
- Build equivalence classes for provably-equal expressions
- Gather information about join ordering restrictions
- Remove useless joins
- ...

Distribute WHERE and JOIN/ON qual clauses

- In general, we want to use each qual at the lowest possible join level
- When dealing with inner joins, we can push a qual down to its "natural" semantic level
- When dealing with outer joins, a qual may be delayed and cannot be pushed down to its "natural" semantic level
- We mark the outerjoin-delayed qual with a "required_relids" including all the required rels in the outer join

Quals that are outerjoin-delayed

- An outer join's own JOIN/ON quals mentioning nonnullable side rels cannot be pushed down below the outer join

```
# EXPLAIN (COSTS OFF) SELECT * FROM foo LEFT JOIN bar ON foo.a = 42;
         QUERY PLAN
----------------------------
 Nested Loop Left Join
   Join Filter: (foo.a = 42)
   ->  Seq Scan on foo
   ->  Materialize
         ->  Seq Scan on bar
(5 rows)
```

Quals that are outerjoin-delayed

- Quals appearing in WHERE or in a JOIN above the outer join cannot be pushed down below the outer join, if they reference any nullable Vars

```
# EXPLAIN (COSTS OFF) SELECT * FROM foo LEFT JOIN bar ON foo.a = bar.c WHERE
COALESCE(bar.c, 1) = 42;
             QUERY PLAN
------------------------------------
 Hash Left Join
   Hash Cond: (foo.a = bar.c)
   Filter: (COALESCE(bar.c, 1) = 42)
   ->  Seq Scan on foo
   ->  Hash
         ->  Seq Scan on bar
(6 rows)
```

EquivalenceClasses

- For mergejoinable equality clauses A = B that are not outerjoin-delayed, we use EquivalenceClasses to record this knowledge
- An EquivalenceClass represents a set of values that are known all transitively equal to each other
- Equivalence clauses are removed from the standard qual distribution process. Instead, eclass-based qual clauses are generated dynamically when needed
- EquivalenceClasses also represent the value that a PathKey orders by (since if x = y, then ORDER BY x must be the same as ORDER BY y)

Gather information about join ordering restrictions

- One-sided outer joins constrain the order of joining partially but not completely
  - non-FULL joins can be freely associated into the lefthand side of an OJ, but in some cases they can't be associated into the righthand side

```
(A leftjoin B on (Pab)) innerjoin C on (Pac)
= (A innerjoin C on (Pac)) leftjoin B on (Pab)
```

```
(A leftjoin B on (Pab)) innerjoin C on (Pbc)
!= A leftjoin (B innerjoin C on (Pbc)) on (Pab)
```

Gather information about join ordering restrictions

- One-sided outer joins constrain the order of joining partially but not completely
- We flatten non-FULL joins to top-level "joinlist" so that they participate fully in the join order search
- We record information about each outer join, in order to avoid generating illegal join orders

Remove useless joins

- A left join can be removed if:
  - innerrel is a single baserel
  - innerrel attributes are not used above the join
  - the join condition cannot match more than one inner-side row

```
SELECT foo.a FROM foo LEFT JOIN (SELECT DISTINCT c AS c FROM bar) sub ON foo.a = sub.c;

=>

SELECT foo.a FROM foo;
```

Scan/join planning

- Basically deals with the FROM and WHERE parts of the query
- Knows about ORDER BY too
  - mainly so that it can design merge-join plans
  - but also to avoid final sort if possible

```
# EXPLAIN (COSTS OFF) SELECT * FROM foo JOIN bar ON foo.a = bar.c AND foo.b = bar.d ORDER BY b, a;
                     QUERY PLAN
-----------------------------------------------------
 Merge Join
   Merge Cond: ((foo.b = bar.d) AND (foo.a = bar.c))
   ->  Sort
         Sort Key: foo.b, foo.a
         ->  Seq Scan on foo
   ->  Sort
         Sort Key: bar.d, bar.c
         ->  Seq Scan on bar
(8 rows)
```

Scan/join planning

- Basically deals with the FROM and WHERE parts of the query
- Knows about ORDER BY too
  - mainly so that it can design merge-join plans
  - but also to avoid final sort if possible
- Cost estimate driven

Scan/join planning

- Identify feasible scan methods for base relations, estimate their costs and result sizes
- Search the join-order space, using dynamic programming or heuristic “GEQO” method, to identify feasible plans for join relations
- Honor outer-join ordering constraints
- Produce one or more “Path” data structures

Join searching

- Multi-way joins have to be built up from pairwise joins, because that’s all the executor knows how to do
- For any given pairwise join step, we can identify the best input Paths and join methods via straightforward cost comparisons, resulting in a list of Paths much as for a base relation
- Finding the best ordering of pairwise joins is the hard part

Join searching

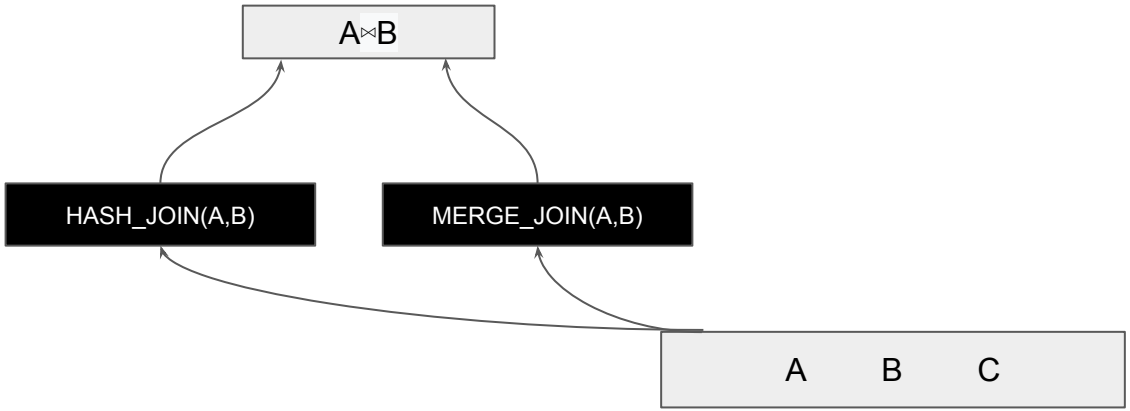
- We usually have many choices of join order for a multi-way join query, and some orders will be cheaper than others
- If the query contains only plain inner joins, we can join the base relations in any order
- Outer joins can be re-ordered in some but not all cases; we handle that by checking whether each proposed join step is legal

Standard join search method

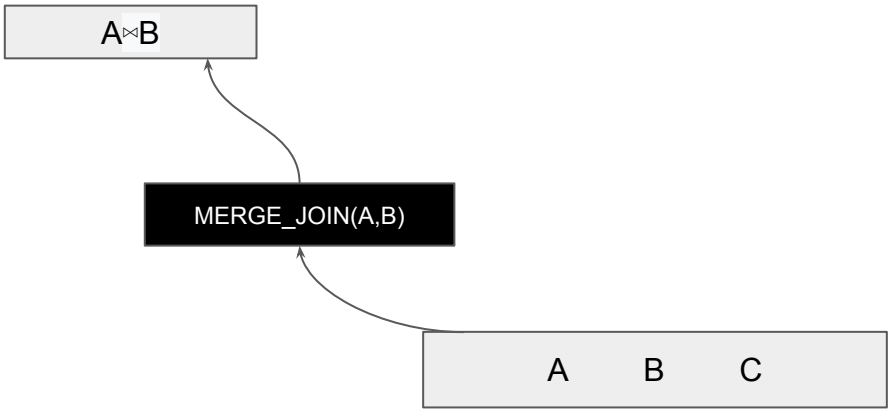
- Generate paths for each base relation
- Generate paths for each possible two-way join
- Generate paths for each possible three-way join
- Generate paths for each possible four-way join
- Continue until all base relations are joined into a single join relation; then use that relation’s best path

```
SELECT * FROM A JOIN (B JOIN C ON B.j = C.j) ON A.i = B.i;
```

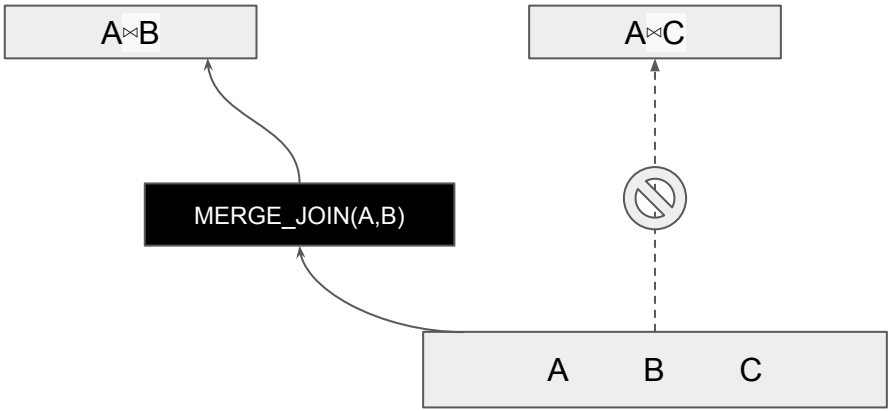
```
SELECT * FROM A JOIN (B JOIN C ON B.j = C.j) ON A.i = B.i;
```



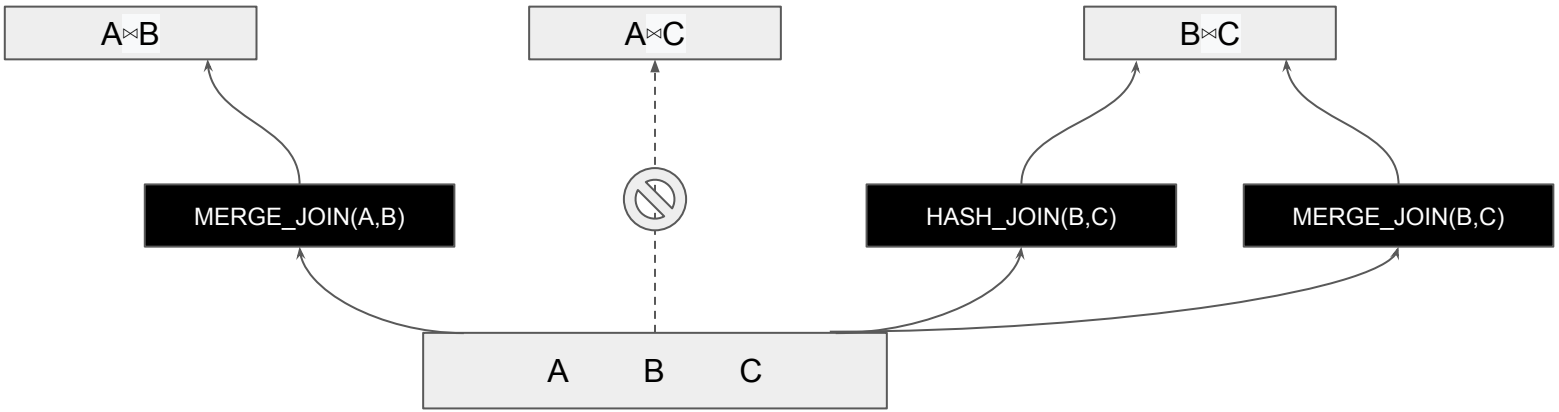
```
SELECT * FROM A JOIN (B JOIN C ON B.j = C.j) ON A.i = B.i;
```



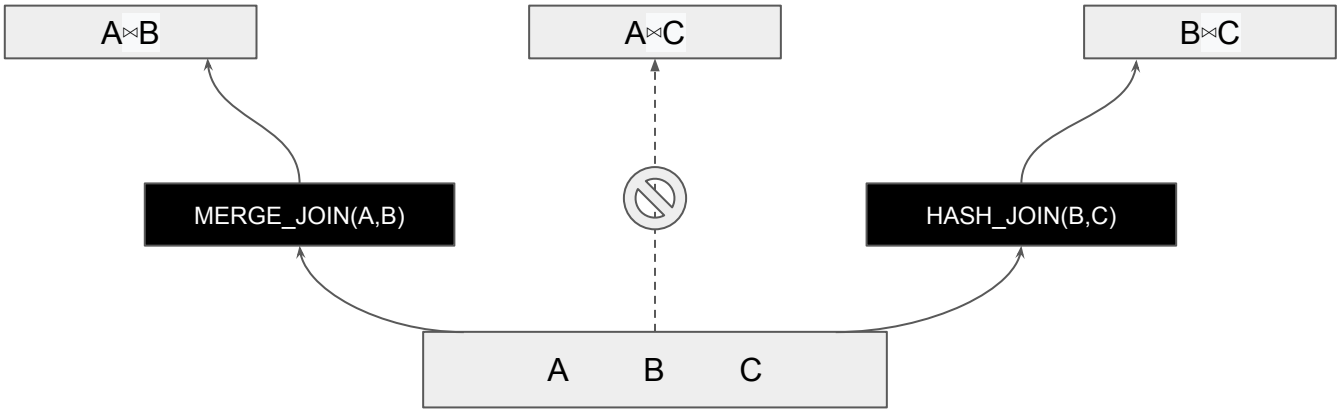
```
SELECT * FROM A JOIN (B JOIN C ON B.j = C.j) ON A.i = B.i;
```



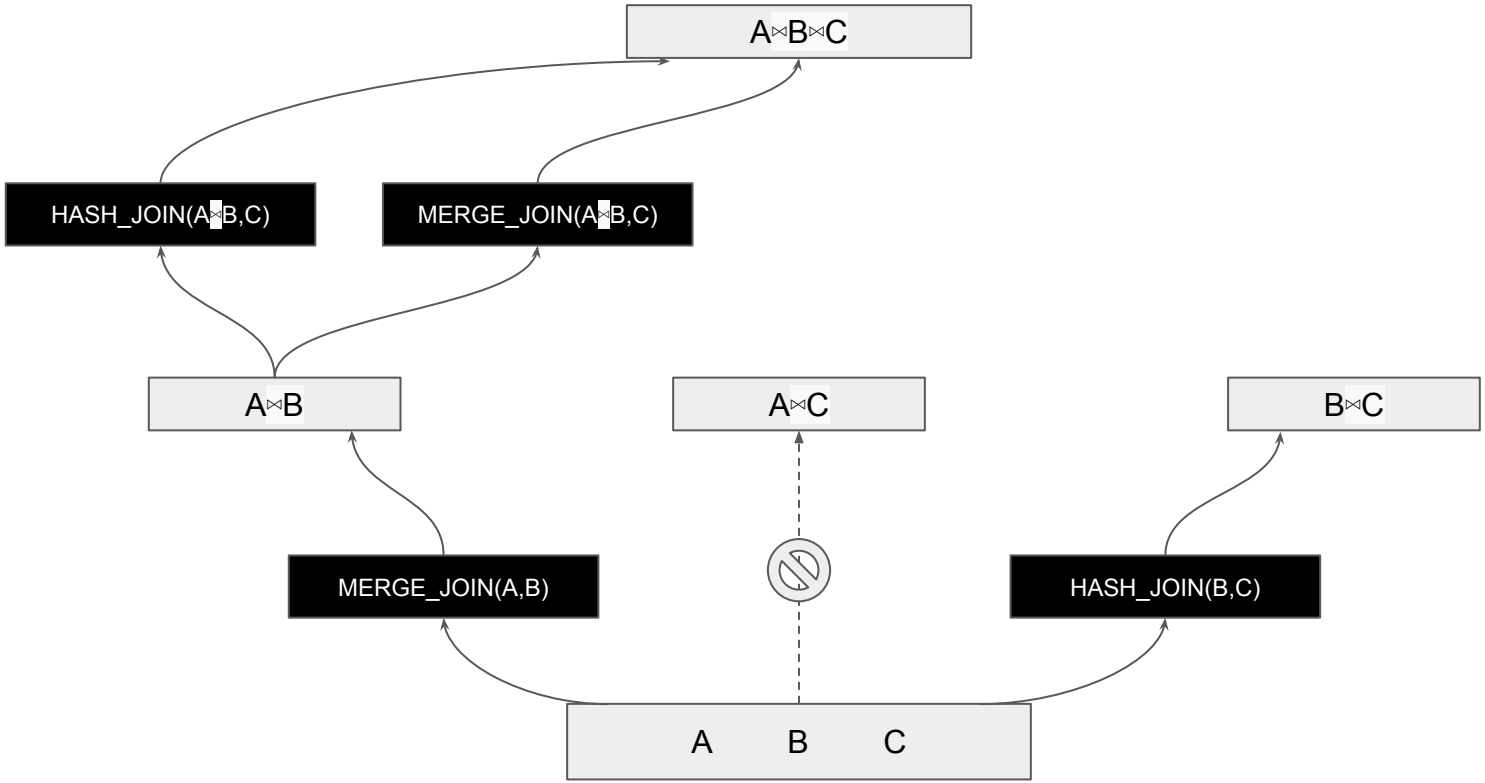
```
SELECT * FROM A JOIN (B JOIN C ON B.j = C.j) ON A.i = B.i;
```



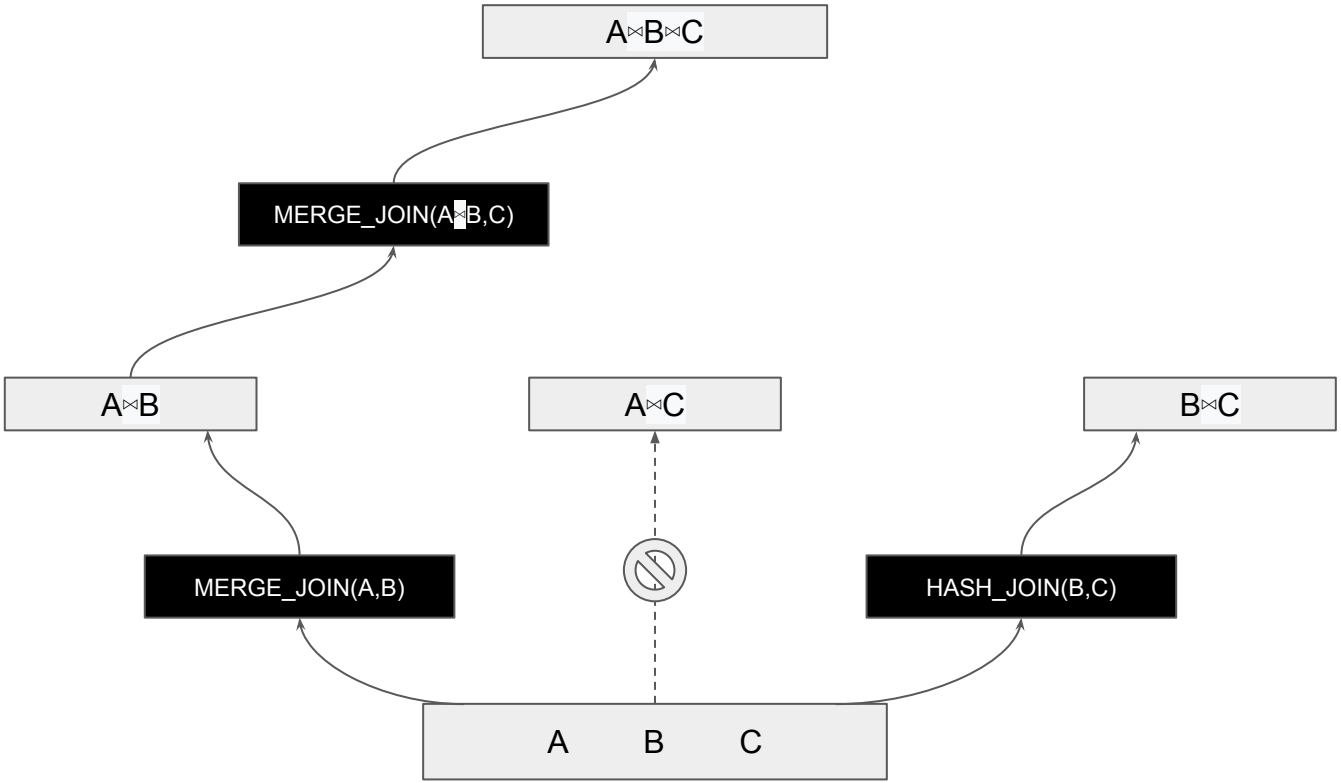
```
SELECT * FROM A JOIN (B JOIN C ON B.j = C.j) ON A.i = B.i;
```



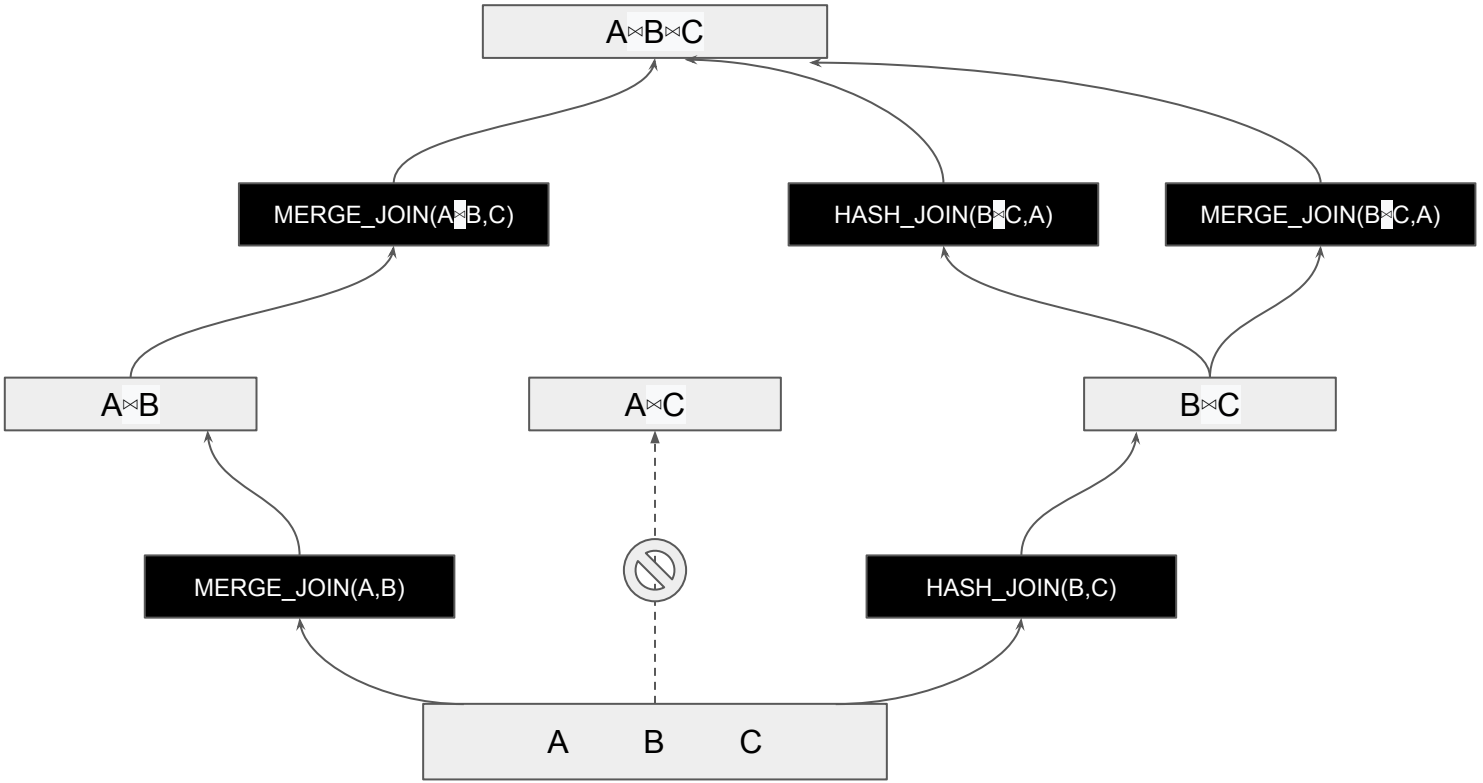
```
SELECT * FROM A JOIN (B JOIN C ON B.j = C.j) ON A.i = B.i;
```



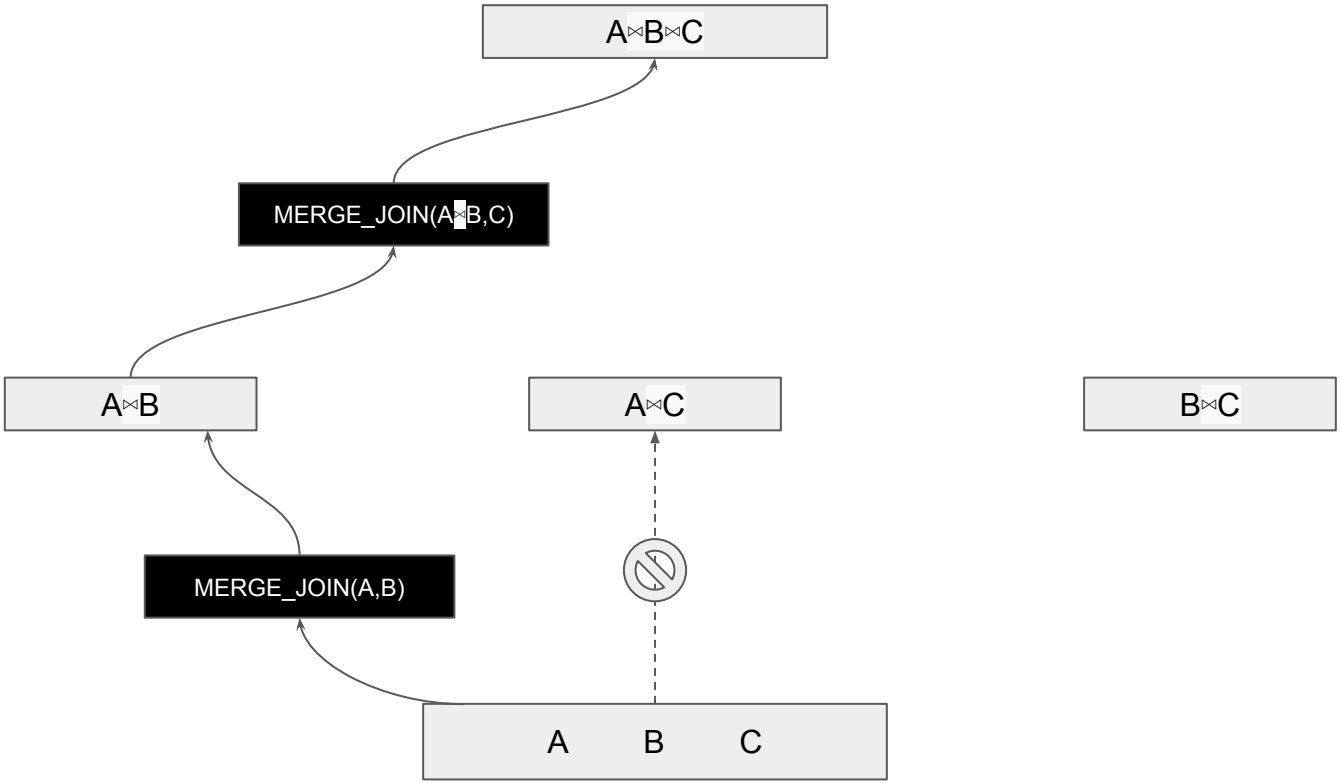
```
SELECT * FROM A JOIN (B JOIN C ON B.j = C.j) ON A.i = B.i;
```



```
SELECT * FROM A JOIN (B JOIN C ON B.j = C.j) ON A.i = B.i;
```



```
SELECT * FROM A JOIN (B JOIN C ON B.j = C.j) ON A.i = B.i;
```



Join searching is expensive

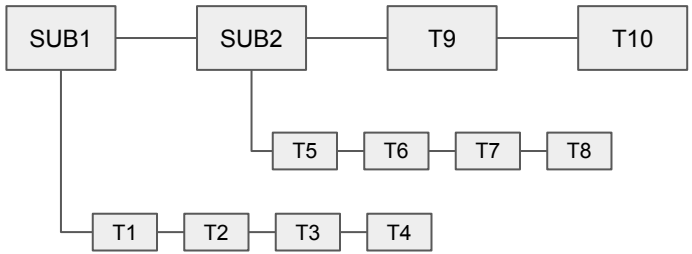
- An **n**-way join problem can potentially be implemented in **n!** (n factorial) different join orders
- It is not feasible to consider all possibilities
- We use a few heuristics, like not considering clause-less joins
- With too many relations (12 by default), fall back to “GEQO” (genetic query optimizer) search

Heuristics used in join search

- Don't join relations that are not connected by any join clause, unless forced to by join-order restrictions
- Break down large join problems into sub-problems by not flattening JOIN clauses according to collapse limit

```
SELECT * FROM
    (SELECT * FROM T1, T2, T3, T4) SUB1 JOIN
    (SELECT * FROM T5, T6, T7, T8) SUB2 ON TRUE JOIN
    (SELECT * FROM T9, T10) SUB3 ON TRUE;

SET join_collapse_limit TO 4;
```



Post scan/join planning

- Deal with GROUP BY, aggregation, window functions, DISTINCT
- Deal with UNION/INTERSECT/EXCEPT
- Apply final sort if needed for ORDER BY
- Produce one or more “Path” data structures for each step
- Add LockRows, Limit, ModifyTable steps to each surviving Path

Postprocessing

- Expand best Path to Plan
- Adjust some representational details of Plan
  - Flatten subquery rangetables into a single list
  - Label Vars in upper plan nodes as OUTER_VAR or INNER_VAR, to refer to the outputs of their subplans
  - Remove unnecessary SubqueryScan, Append, and MergeAppend plan nodes
  - etc.

Thank You

Output: Thank You

```
Index Scan using common_phrases_idx on common_phrases
  Index Cond: (value = 'Thank You'::text)
  Filter: (language = 'English'::text)
```